

Sound Static Deadlock Analysis for C/Pthreads (Extended Version)

Daniel Kroening
University of Oxford
Oxford, UK
kroening@cs.ox.ac.uk

Daniel Poetzl
University of Oxford
Oxford, UK
daniel.poetzl@cs.ox.ac.uk

Peter Schrammel
University of Sussex
Brighton, UK
p.schrammel@sussex.ac.uk

Björn Wachter
University of Oxford
Oxford, UK
bjoern.wachter@gmail.com

ABSTRACT

We present a static deadlock analysis approach for C/pthreads. The design of our method has been guided by the requirement to analyse real-world code. Our approach is sound (i.e., misses no deadlocks) for programs that have defined behaviour according to the C standard, and precise enough to prove deadlock-freedom for a large number of programs. The method consists of a pipeline of several analyses that build on a new context- and thread-sensitive abstract interpretation framework. We further present a lightweight dependency analysis to identify statements relevant to deadlock analysis and thus speed up the overall analysis. In our experimental evaluation, we succeeded to prove deadlock-freedom for 262 programs from the Debian GNU/Linux distribution with in total 2.6 MLOC in less than 11 hours.

CCS Concepts

•Theory of computation → Program analysis;

Keywords

deadlock detection, lock analysis, static analysis, abstract interpretation

1. INTRODUCTION

Locks are the most frequently used synchronisation mechanism in concurrent programs to guarantee atomicity, prevent undefined behaviour, and hide weak-memory effects of the underlying architectures. However, locks, if not correctly used, can cause a *deadlock*, where one thread holds a lock that the other one needs and vice versa. Deadlocks can have disastrous consequences such as the Northeastern Blackout [34], where a monitoring system froze, and prevented detection of a local power problem, which brought down the power network of the Northeastern United States.

In small programs, deadlocks may be spotted easily. However, this is not the case in larger software systems. So called *locking disciplines* aim at preventing deadlocks but are difficult to maintain as the system evolves, and every extension bears the risk of introducing deadlocks. For example, if the order in which locks are acquired is different in different parts of the code this may cause a deadlock.

The problem is exacerbated by the fact that deadlocks are difficult to discover by means of testing. Even a test suite with full line coverage is insufficient to detect all deadlocks, and similar to other concurrency bugs, triggering a deadlock requires a specific thread schedule and set of particular program inputs. Therefore, static analysis is a promising candidate for a thorough check for deadlocks. The challenge is to devise a method that is scalable and yet precise enough. The inherent trade-off between scalability and precision has gated static approaches to deadlock analysis, and many previously published results had to resort to unsound approximations in order to obtain scalability (we provide a comprehensive survey of the existing static approaches in Sec. 9).

We hypothesise that static deadlock detection can be performed with a sufficient degree of precision and scalability and without sacrificing soundness. To this end, this paper presents a new method for statically detecting deadlocks. To quantify scalability, we have applied our implementation to a large body of real-world concurrent code from the Debian GNU/Linux project.

Specifically, this paper makes the following contributions:

1. The first, to the best of our knowledge, *sound* static deadlock analysis approach for C/pthreads that can handle real-world code.
2. A new context- and thread-sensitive abstract interpretation framework that forms the basis for the analyses that comprise our approach. The framework unifies contexts, threads, and program locations via the concept of a place.
3. A novel lightweight dependency analysis which identifies statements that could affect a given set of program expressions. We use it to speed up the pointer analysis by focusing it to statements that are relevant to deadlock analysis.
4. We show how to build a lock graph that soundly captures a variety of sources of imprecision, such as may-point-to information and thread creation in loops/recursions, and how to combine the cycle detection with

```

1 int main()
2 {
3     pthread_t tid;
4
5     pthread_create(&tid, 0,
6         thread, 0);
7
8     pthread_mutex_lock(&m1);
9     pthread_mutex_lock(&m3);
10    pthread_mutex_lock(&m2);
11    func1();
12    pthread_mutex_unlock(&m2);
13    pthread_mutex_unlock(&m3);
14    pthread_mutex_unlock(&m1);
15
16    pthread_join(tid, 0);
17
18    int r;
19    r = func2(5);
20
21    return 0;
22 }
23
24 void func1()
25 {
26     x = 0;
27 }
28
29 void *thread()
30 {
31     pthread_mutex_lock(&m1);
32     pthread_mutex_lock(&m2);
33     pthread_mutex_lock(&m3);
34     x = 1;
35     pthread_mutex_unlock(&m3);
36     pthread_mutex_unlock(&m2);
37     pthread_mutex_unlock(&m1);
38
39     pthread_mutex_lock(&m4);
40     pthread_mutex_lock(&m5);
41     x = 2;
42     pthread_mutex_unlock(&m5);
43     pthread_mutex_unlock(&m4);
44
45     return 0;
46 }
47
48 int func2(int a)
49 {
50     pthread_mutex_lock(&m5);
51     pthread_mutex_lock(&m4);
52     if (a)
53         x = 3;
54     else
55         x = 4;
56     pthread_mutex_unlock(&m4);
57     pthread_mutex_unlock(&m5);
58     return 0;
59 }

```

Figure 4: Example of a deadlock-free program

a non-concurrency check to prune infeasible cycles.

5. A thorough experimental evaluation on 715 programs from Debian GNU/Linux with 7.1 MLOC in total and up to 50KLOC per program.

2. OVERVIEW

The design of our analyses has been guided by the goal to analyse real-world concurrent C/threads code in a *sound* way. Fig. 3 gives an overview of our analysis pipeline. An arrow between two analyses indicates that the target uses information computed by the source. We use a dashed arrow from the non-concurrency analysis to the cycle detection to indicate that the required information is computed on-demand (i.e., the cycle detection may repeatedly query the non-concurrency analysis, which computes the result in a lazy fashion). All of the analyses operate on a graph representation of the program (introduced in Sec. 3.1). The exception is the cycle detection phase, which only uses the lock graph computed in the lock graph construction phase.

The pointer analysis, may- and must-lockset analysis, and the lock graph construction are implemented on top of our new generic context- and thread-sensitive analysis framework (described in detail in Sec. 3.2). To enable trade-off between precision and cost, the framework comes in a flow-insensitive and a flow-sensitive version. The pointer analysis was implemented on top of the former (thus marked with ****** in Fig. 3), and the may- and must-lockset analysis and the lock graph construction on top of the latter (marked with *****). The dependency analysis and the non-concurrency analysis are separate standalone analyses.

Context and thread sensitivity.

Typical patterns in real-world C code suggest that an approach that provides a form of context-sensitivity is necessary to obtain satisfactory precision on real-world code, as otherwise there would be too many false deadlock reports. For instance, many projects provide their own wrap-

pers for the functions of the pthreads API. Fig. 1, for example, shows a lock wrapper from the VLC project. An analysis that is not context-sensitive would merge the points-to information for pointer **p** from different call sites invoking **vlc_mutex_lock()**, and thus yield many false alarms.

Thread creation causes a similar problem. For every call to **pthread_create()**, the analysis needs to determine which thread is created (i.e., the function identified by the pointer passed to **pthread_create()**). This is straightforward if a function identifier is given to **pthread_create()**. However, similar to the case of lock wrappers above, projects often provide wrappers for **pthread_create()**. Fig. 1 shows a wrapper for **pthread_create()** from the memcached project. The wrapper then uses the function pointer that is passed to **create_worker()** to create a thread. Maintaining precision in such cases requires us to track the flow of function pointer values from function arguments to function parameters. This is implemented directly as part of the analysis framework (as opposed to in the full points-to analysis).

Dependency analysis.

Deadlock detection requires the information which lock objects an expression used in a **pthread_mutex_lock()** call may refer to. We compute this data using the pointer analysis, which is potentially expensive. However, it is easy to see that potentially many assignments and function calls in a program do not affect the values of lock expressions. Consider for example Fig. 4. The accesses to **x** cannot affect the value of the lock pointers **m1–m5**. Further, the code in function **func1** cannot affect the values of the lock pointers, and thus in turn the call **func1()** in line 11 cannot affect the lock pointers.

We have developed a lightweight context-insensitive, flow-insensitive analysis to identify statements that may affect a given set of expressions. The result is used to speed up the pointer analysis. The dependency analysis is based on marking statements which (transitively) share common variables with the given set of expressions. In our case, the relevant expressions are those used in lock-, create-, and join-statements. For the latter two we track the thread ID variable (first parameter of both) whose value is required to determine which thread is joined by a join operation. We give the details of the dependency analysis in Sec. 4.

Non-concurrency analysis.

A deadlock resulting from a thread 1 first acquiring lock m_1 and then attempting to acquire m_2 (at program location ℓ_1), and thread 2 first acquiring m_2 and then attempting to acquire m_1 (at program location ℓ_2) can only occur when in a concrete program execution the program locations ℓ_1 and ℓ_2 run concurrently. If we have a way of deciding whether two locations could potentially run concurrently, we can use this information to prune spurious deadlock reports. For this purpose we have developed a non-concurrency analysis that can detect whether two statements cannot run concurrently based on two criteria.

Common locks. If thread 1 and thread 2 hold a common lock at locations ℓ_1 and ℓ_2 , then they cannot both simultaneously reach those locations, and hence the deadlock cannot happen. This is illustrated in Fig. 4. The thread **main()** attempts to acquire the locks in the sequence m_1, m_3, m_2 , and the thread **thread()** attempts to acquire the locks in the sequence m_1, m_2, m_3 . There is an order inversion between m_2 and m_3 , but there is no deadlock since the two sections 8–14 and 28–34 (and thus in particular the locations 10 and 30) are protected by the common lock m_1 . The common locks

```

1 void vlc_mutex_lock (vlc_mutex_t *p) {
2   int val = pthread_mutex_lock(p);
3   VLC_THREAD_ASSERT("locking_mutex");
4 }
5
6 void create_worker(void *(*func)(void *),
7                  void *arg) {
8   pthread_attr_t attr;
9   int ret;
10  pthread_attr_init(&attr);
11  if ((ret=pthread_create(
12    &((THREAD*)arg)->thread_id,
13    &attr, func, arg)) != 0) {
14    fprintf(stderr, "Error: %s\n",
15            strerror(ret));
16    exit(1);
17  }
18 }

```

Figure 1: Lock and create wrappers

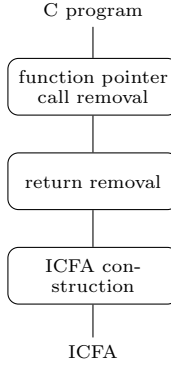


Figure 2: ICFA constr.

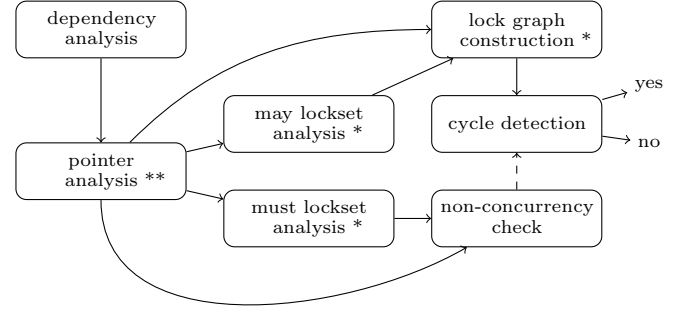


Figure 3: Analysis pipeline

criterion has first been described by Havelund [18] (common locks are called *gatelocks* there).

Create and join. Statements might also not be able to run concurrently because of the relationship between threads due to the `pthread_create()` and `pthread_join()` operations. In Fig. 4, there is an order inversion between the locks of m_5 and m_4 by function `func2()`, and the locks of m_4 , m_5 of thread `thread()`. Yet there is no deadlock since the thread `thread()` is joined before `func2()` is invoked.

Our non-concurrency analysis makes use of the must lockset analysis (computing the locks that must be held) to detect common locks. To detect the relationship between threads due to create and join operations it uses a search on the program graph for joins matching earlier creates. We give more details of our non-concurrency analysis in Sec. 5.

3. ANALYSIS FRAMEWORK

In this section, we first introduce our program representation, then describe our context- and thread-sensitive framework, and then describe the pointer analysis and lockset analyses that are implemented on top of the framework.

3.1 Program Representation

Preprocessing. Our tool takes as input a concurrent C program using the pthreads threading library. In a first step the calls to functions through function pointers are removed. A call is replaced by a case distinction over the functions the function pointer could refer to. Specifically, a function pointer can only refer to functions that are type-compatible and of which the address has been taken at some point in the code. This is illustrated in Fig. 5 (top). Functions `f2()` (address not taken) and `f4()` (not type-compatible) do not have to be part of the case distinction. In a second step, functions with multiple exit points (i.e., multiple return statements) are transformed such as to have only one exit points (illustrated in Fig. 5 (bottom)).

Interprocedural CFAs. We transform the program into a graph representation which we term *interprocedural control flow automaton (ICFA)*. The functions of the program are represented as CFAs [19]. CFAs are similar to control flow graphs, but with the nodes representing program locations and the edges being labeled with operations. ICFA have additional inter-function edges modeling function entry, function exit, thread entry, thread exit, and thread join. Fig. 4 shows a concurrent C program and Fig. 6 shows its corresponding ICFA (leaving off thread exit and thread join edges and the function `func1()`).

We denote by *Prog* a program (represented as an ICFA), by *Funcs* the set of identifiers of the functions, by $L = \{\ell_0, \dots, \ell_{n-1}\}$ the set of program locations, by E the set of edges connecting the locations, and by $\text{op}(e)$ a function that labels each edge with an operation. For example, in Fig. 6, the edge between locations 49 and 52 is labeled with the operation `x=3`, and the edge between locations 19 and 46 is labeled with the operation `func_entry(5, a)`.

We further write $L(f)$ for the locations in function f . Each program location is contained in exactly one function. The function `func(ℓ)` yields the function that contains ℓ . The set of variable identifiers in the program is denoted by *Vars*. We assume that all identifiers in *Prog* are unique, which can always be achieved by a suitable renaming of identifiers.

We treat lock, unlock, thread create, and thread join as primitive operations. That is, we do not analyse the body of e.g. `pthread_create()` (as implemented in e.g. glibc on GNU/Linux systems). Instead, our analysis only tracks the semantic effect of the operation, i.e., creating a new thread.

Apart from intra-function edges we also have inter-function edges that can be labeled with the five operations `func_entry`, `func_exit`, `thread_entry`, `thread_exit`, and `thread_join`.

A function entry edge (`func_entry`) connects a call site to the function entry point. The edge label also includes the function call arguments and the function parameters. For example, `func_entry(5, a)` indicates that the integer literal 5 is passed to the call as an argument, which is assigned to function parameter a . A function exit edge (`func_exit`) connects the exit point of a function to every call site calling the function. Our analysis algorithm filters out infeasible edges during the exploration of the ICFA. That is, if a function entry edge is followed from a function f_1 to function f_2 , then the analysis algorithm later follows the exit edge from f_2 to f_1 , disregarding exit edges to other functions.

A thread entry edge (`thread_entry`) connects a thread creation site to all potential thread entry points. It is necessary to connect to all potential thread entry points since often a thread creation site can create threads of different types (i.e., corresponding to different functions), depending on the value of the function pointer passed to `pthread_create()`. Analogous to the case of function exit edges, our analysis algorithm tracks the values of function pointers during the ICFA exploration. At a thread creation site it thus can resolve the function pointer, and only follows the edge to the thread entry point corresponding to the value of the function pointer.

A `thread_exit` edge connects the exit point of a thread to the location following all thread creation sites, and a

```

1 void f1() {}
2 void f2() {}
3 void f3() {}
4 int f4() {}
5 ...
6 ... = &f1;
7 ... = &f3;
8 ... = &f4;
9 fp();

```

⇒

```

1 int f() {
2   if (...)
3     return 0;
4   else
5     return 1;
6 }
7 ...
8 a = f();

```

⇒

```

1 int f() {
2   int ret;
3   if (...)
4     ret = 0;
5   goto END;
6   else
7     ret = 1;
8   goto END;
9 END:
10  return ret;
11 }
12 ...
13 a = f();

```

Figure 5: Function pointers and returns

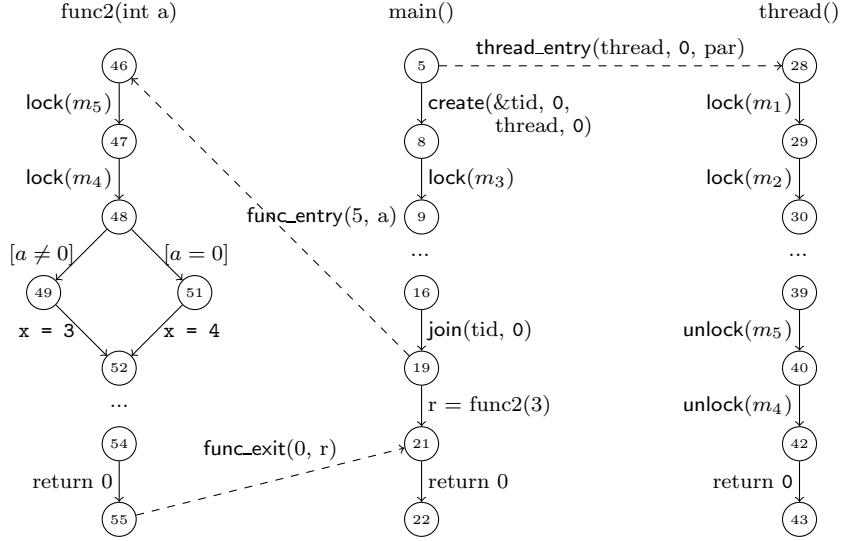


Figure 6: ICFA associated with the program in Fig. 4

`thread_join` edge connects a thread exit point to all join operations in the program.

3.2 Analysis Framework – Overview

Our framework to perform context- and thread-sensitive analyses on ICFA is based on abstract interpretation [12]. It implements a flow-sensitive and flow-insensitive fixpoint computation over the ICFA, and needs to be parametrised with a custom analysis to which it delegates the handling of individual edges of the ICFA. We provide more details and a formalization of the framework in the next section.

Our analysis framework unifies contexts, threads, and program locations via the concept of a *place*. A place is a tuple $(\ell_0, \ell_1, \dots, \ell_n)$ of program locations. The program locations $\ell_0, \dots, \ell_{n-1}$ are either function call sites or thread creation sites in the program (such as, e.g., location 19 in Fig. 6). The final location ℓ_n can be a program location of any type. The locations $\ell_0, \dots, \ell_{n-1}$ model a possible function call and thread creation history that leads up to program location ℓ_n . We denote the set of all places for a given program by P . We use the $+$ operator to extend tuples, i.e., $(\ell_0, \dots, \ell_{n-1}) + \ell_n = (\ell_0, \dots, \ell_{n-1}, \ell_n)$. We further write $|p|$ for the length of the place. We write $p[i]$ for element i (indices are 0-based). We use slice notation to refer to contiguous parts of places; $p[i:j]$ denotes the part from index i (inclusive) to index j (exclusive), and $p[:i]$ denotes the prefix until index i (exclusive). We write $\text{top}(p)$ for the last location in the place.

As an example, in Fig. 6, place $(19, 49)$ denotes the program location 49 in function `func2()` when it has been invoked at call site 19 in the main function. If function `func2()` were called at multiple program locations ℓ_1, \dots, ℓ_m in the main function, we would have different places $(\ell_1, 49), \dots, (\ell_m, 49)$ for location 49 in function `func2()`. Similarly, for the thread function `thread()` and, e.g., location 29, we have a place $(5, 29)$ with 5 identifying the creation site of the thread.

Each place has an associated abstract thread identifier, which we refer to as *thread ID* for short. Given a place $p = (\ell_0, \dots, \ell_n)$, the associated thread ID is either $t = ()$ (the empty tuple) if no location in p corresponds to a thread creation site, or $t = \ell_0, \dots, \ell_i$, such that ℓ_i is a thread cre-

ation site and all ℓ_j with $j > i$ are not thread creation sites. It is in this sense that our analysis is thread-sensitive, as the information computed for each place can be associated with an abstract thread that way. We write $\text{get_thread}(p)$ for the thread ID associated with place p .

The analysis framework must be parametrised with a custom analysis. The framework handles the tracking of places, the tracking of the flow of function pointer values from function arguments to function parameters, and it invokes the custom analysis to compute dataflow facts for each place.

The domain, transfer function, and join function of the framework are denoted by \mathcal{D}_s , \mathcal{T}_s , and \sqcup_s , respectively, and the domain, transfer function, and join function of the parametrising analysis are denoted by \mathcal{D}_a , \mathcal{T}_a , and \sqcup_a . The custom analysis has a transfer function $\mathcal{T}_a : E \times P \rightarrow (\mathcal{D}_a \rightarrow \mathcal{D}_a)$ and a join function $\sqcup_a : \mathcal{D}_a \times \mathcal{D}_a \rightarrow \mathcal{D}_a$. The domain of the framework (parametrised by the custom analysis) is then $\mathcal{D}_s = \text{Fpms} \times \mathcal{D}_a$, the transfer function is $\mathcal{T}_s : E \times P \rightarrow (\mathcal{D}_s \rightarrow \mathcal{D}_s)$, and the join function is $\sqcup_s : \mathcal{D}_s \times \mathcal{D}_s \rightarrow \mathcal{D}_s$.

The set Fpms is a set of mappings from identifiers to functions which map function pointers to the functions they point to. We denote the empty mapping by \emptyset . We further write $\text{fpm}(fp) = \perp$ to indicate that fp is not in $\text{dom}(\text{fpm})$ (the domain of fpm). A function pointer fp might be mapped by fpm either to a function f or to the special value d (for “dirty”) which indicates that the analysed function assigned to fp or took the address of fp . In this case we conservatively assume that the function pointer could point to any thread function.

3.3 Analysis Framework – Details

We now describe the formalization of the analysis framework which is shown in Fig. 7. The figure gives the flow-sensitive variant of our framework. We refer to Appendix A for the flow-insensitive version. The figure gives the domain, join function \sqcup_s and transfer function \mathcal{T}_s which are defined in terms of the join function \sqcup_a and transfer function \mathcal{T}_a of the parametrising analysis (such as the lockset analyses defined in the next section).

The function $\text{next}_s(e, p)$ defines how the place p is updated when the analysis follows the ICFA edge e . For example, on

Domain: $\mathcal{D}_s = Fpms \times \mathcal{D}_a$	
$s_s^1 \sqcup_s s_s^2 = (fpm, s_a)$ $\text{with } s_s^1 = (fpm^1, s_a^1)$ $\text{with } s_s^2 = (fpm^2, s_a^2)$ $\text{with } fpm = fpm^1 \sqcup_{fp} fpm^2$ $\text{with } s_a = s_a^1 \sqcup_a s_a^2$ $fpm^1 \sqcup_{fp} fpm^2 = fpm$	
$fpm(fp) =$ $\begin{cases} d & fpm^1(fp) = d \vee fpm^2(fp) = d \\ v & (fpm^1(fp) = v \wedge (fp \notin \text{dom}(fpm^2) \vee fpm^2(fp) = v)) \vee \\ & (fpm^2(fp) = v \wedge (fp \notin \text{dom}(fpm^1) \vee fpm^1(fp) = v)) \\ \perp & \text{otherwise} \end{cases}$	
With $e = (\ell_1, \ell_2)$, $\text{top}(p) = \ell_1$, $f = \text{func}(\ell_2)$, and $n = p $:	
$\text{next}_s(e, p) =$ $\begin{cases} \text{entry}_s(p, \ell_2) & \text{op}(e) \in \{\text{thread_entry}, \text{func_entry}\} \\ p[:n-2] + \ell_2 & \text{op}(e) \in \{\text{func_exit}, \text{thread_exit}, \text{thread_join}\} \\ p[:n-1] + \ell_2 & \text{otherwise} \end{cases}$	
$\text{entry}_s(p, \ell) =$ $\begin{cases} p' + \ell' + \ell & p = p' + \ell' + \ell'' + p'' \wedge \text{func}(\ell'') = f \\ p + \ell & \text{otherwise} \end{cases}$	
With $e = (\ell_{src}, \ell_{tgt})$, $\text{op}(e) = \text{func_entry}(arg_1, \dots, arg_k, par_1, \dots, par_k)$, and $s_s = (fpm, s_a)$:	
$\mathcal{T}_s[e, p](s_s) = (fpm', \mathcal{T}_a[e, p](s_a))$ $fpm'(par_i) = \begin{cases} arg_i & \text{is_func}(arg_i) \\ fpm(arg_i) & \text{is_func_pointer}(arg_i) \end{cases}$	
With $e = (\ell_{src}, \ell_{tgt})$, $f = \text{func}(\ell_{tgt})$, $\text{op}(e) = \text{thread_entry}(thr, arg, par)$, and $s_s = (fpm, s_a)$:	
$\mathcal{T}_s[e, p](s_s) = \begin{cases} (fpm', \mathcal{T}_a[e, p](s_a)) & \text{match_fp}(fpm, thr, f) \\ (\emptyset, \perp_a) & \text{otherwise} \end{cases}$ $\text{match_fp}(fpm, thr, f) =$ $(\text{is_func_pointer}(thr) \wedge fpm(thr) \in \{\perp, d, f\}) \vee$ $(\text{is_func}(thr) \wedge thr = f)$	
With $e = (\ell_{src}, \ell_{tgt})$, $\text{op}(e) \in \{\text{func_exit}, \text{thread_exit}, \text{thread_join}\}$, and $s_s = (fpm, s_a)$:	
$\mathcal{T}_s[e, p](s_s) = (\emptyset, \mathcal{T}_a[e, p](s_a))$	
With $e = (\ell_{src}, \ell_{tgt})$, $\text{op}(e) = \text{op}$, and $s_s = (fpm, s_a)$:	
$\mathcal{T}_s[e, p](s_s) = (fpm, \mathcal{T}_a[e, p](s_a))$	

Figure 7: Context-, thread-, and flow-sensitive framework

a **func_exit** edge, the last two locations are removed from the place (which are the exit point of the function, and the location of the call to the function), and the location to which

the function returns to is added to the place (which is the location following the call to the function). The **thread_entry** and **func_entry** cases are delegated to $\text{entry}_s(p, \ell)$. The first case of the function handles recursion. If a location ℓ'' of the called function is already part of the place, then the prefix of the place that corresponds to the original call to the function is reused (first case). If no recursion is detected, the entry location of the function is simply added to the current place (second case). For intra-function edges (last case of next_s), the last location is removed from the place and the target location of the edge is added.

The overall result of the analysis is a mapping $s \in P \rightarrow (Fpms \times \mathcal{D}_a)$. The result is defined via a fixpoint equation [12]. We obtain the result by computing the least fixpoint (via a worklist algorithm) of the equation below (with s_0 denoting the initial state of the places):

$$s = s_0 \sqcup \lambda p. \bigcup_{p', e \text{ s.t. } np(p, p', e)} \mathcal{T}_s[e, p'](s(p'))$$

$$\text{with } np(p, p', (\ell_1, \ell_2)) = \ell_1 = \text{top}(p') \wedge$$

$$\ell_2 = \text{top}(p) \wedge$$

$$\text{next}_s((\ell_1, \ell_2), p') = p$$

$$\text{with } s \sqcup s' = \lambda p. s(p) \sqcup_c s'(p)$$

The equation involves computing the join over all places p' and edges e in the ICFA such that $np(p, p', e)$.

We next describe the definition of the transfer function of the framework in more detail. The definition consists of four cases: (1) function entry, (2) thread entry, (3) function exit, thread exit, thread join, and (4) intra-function edges.

(1) When applying a function entry edge, a new function pointer map fpm' is created by assigning arguments to parameters and looking up the values of the arguments in the current function pointer map fpm . As in the following cases, the transfer function \mathcal{T}_a of the custom analysis is applied to the state s_a .

(2) Applying a thread entry edge to a state s_c yields one of two outcomes. When the value of the function pointer argument thr matches the target of the edge (i.e., the edge enters the same function as the function pointer points to), then the function pointer map is updated with arg and par (as in the previous case), and the transfer function of the custom analysis is applied. Otherwise, the result is the bottom element $\perp_c = (\emptyset, \perp_a)$.

(3) The function pointer map is cleared (as its domain contains only parameter identifiers which are not accessible outside of the function), and the custom transfer function is applied.

(4) The custom transfer function is applied.

As is not shown for lack of space in Fig. 7, if a function pointer fp is assigned to or its address is taken, its value is set to d in fpm , thus indicating that it could point to any thread function.

Implementation.

During the analysis we need to keep a mapping from places to abstract states (which we call the *state map*). However, directly using the places as keys for the state maps in all analyses can lead to high memory consumption. Our implementation therefore keeps a global two-way mapping (shared by all analyses in Fig. 3) between places and unique IDs for the places (we call this the *place map*). The state maps of the analyses are then mappings from unique IDs to abstract states, and the analyses consult the global place map

Domain: $2^{Objs} \cup \{\{\star\}\}$
$s_1 \sqcup s_2 = \begin{cases} s_1 \cup s_2 & \text{if } s_1, s_2 \neq \{\star\} \\ \{\star\} & \text{otherwise} \end{cases}$
With $\text{op}(e) = \text{lock}(a)$: $\mathcal{T}[e, p](s) = \begin{cases} s \cup \text{vs}(p, a) & \text{if } s, \text{vs}(p, a) \neq \{\star\} \\ \{\star\} & \text{otherwise} \end{cases}$
With $\text{op}(e) = \text{unlock}(a)$: $\mathcal{T}[e, p](s) = \begin{cases} \emptyset & \text{if } s = 1 \wedge s \neq \{\star\} \\ s - \text{vs}(p, a) & \text{if } s \cap \text{vs}(p, a) = 1 \wedge s \neq \{\star\} \wedge \text{vs}(p, a) \neq \{\star\} \\ s & \text{otherwise} \end{cases}$
With $\text{op}(e) \in \{\text{thread_entry}, \text{thread_exit}, \text{thread_join}\}$: $\mathcal{T}[e, p](s) = \emptyset$

Figure 8: May lockset analysis

to translate between places and IDs when needed.

In the two-way place map, the mapping from places to IDs is implemented via a trie, and the mapping from IDs to places via an array that stores pointers back into the trie. The places in a program can be efficiently stored in a trie as many of them share common prefixes. We give further details in Figure 16 in Appendix D.

3.4 Pointer Analysis

We use a standard points-to analysis that is an instantiation of the flow-insensitive version of the above framework (see Appendix A). It computes for each place an element of $\text{Vars} \rightarrow (2^{Objs} \cup \{\{\star\}\})$. That is, the set of possible values of a pointer variable is either a finite set of objects it may point to, or $\{\star\}$ to indicate that it could point to any object. We use $\text{vs}(p, a)$ to denote the value set at place p of pointer a . The pointer analysis is sound for concurrent programs due to its flow-insensitivity [32].

3.5 Lockset Analysis

Our analysis pipeline includes a may lockset analysis (computing for each place the locks that may be held) and a must lockset analysis (computing for each place the locks that must be held). The former is used by the lock graph analysis, and the latter by the non-concurrency analysis.

The may lockset analysis is formalised in Fig. 8 as a custom analysis to parametrise the flow-sensitive framework with. The must lockset analysis is given in Appendix C. Both the may and must lockset analyses makes use of the previously computed points-to information by means of the function $\text{vs}()$. In both cases, care must be taken to compute sound information from the may-point-to information provided by $\text{vs}()$. For example, for the may lockset analysis on an $\text{unlock}(a)$ operation, we cannot just remove all elements in $\text{vs}(p, a)$ from the lockset, as an unlock can only unlock one lock. We use $ls_a(p), ls_u(p)$ to denote the may and must locksets at place p .

4. DEPENDENCY ANALYSIS

We have developed a context-insensitive, flow-insensitive *dependency analysis* to compute the set of assignments and function calls that might affect the value of a given set of expressions (in our case the expressions used in lock-, create-, and join-statements). The purpose of the analysis is to speed up the following pointer analysis phase (cf. Fig. 3).

Below we first describe a semantic characterisation of dependencies between expressions and assignments, and then devise an algorithm to compute dependencies based on syntax only (specifically, the variable identifiers occurring in the expressions/assignments).

Semantic characterisation of dependencies.

Let $AS = \{e \in E(\text{Prog}) \mid \text{is_assign}(\text{op}(e))\}$ be the set of assignment edges. Let exprs be a set of starting expressions. Let further $R(a), W(a)$ denote the set of memory locations that an expression or assignment a may read (resp. write) over *all* possible executions of the program. Let further $M(a) = R(a) \cup W(a)$. Then we define the immediate dependence relation dep as follows (with $*$ denoting transitive closure and $;$ denoting composition):

$$\begin{aligned} \text{dep}_1 &\subseteq \text{exprs} \times AS, (a, b) \in \text{dep}_1 \Leftrightarrow R(a) \cap W(b) \neq \emptyset \\ \text{dep}_2 &\subseteq AS \times AS, (a, b) \in \text{dep}_2 \Leftrightarrow R(a) \cap W(b) \neq \emptyset \\ \text{dep} &= \text{dep}_1; \text{dep}_2^* \end{aligned}$$

If $(a, b) \in \text{dep}_1$, then the evaluation of expression a may read a memory location that is written to by assignment b . If $(a, b) \in \text{dep}_2$, then the evaluation of the assignment a may read a memory location that is written to by the assignment b . If $(a, b) \in \text{dep}$, this indicates that the expression a can (transitively) be influenced by the assignment b . We say a depends on b in this case.

The goal of our dependency analysis is to compute the set of assignments $A = \text{dep}|_{(_, a) \rightarrow a}$ (the binary relation A projected to the second component). However, we cannot directly implement a procedure based on the definitions above as this would require the functions $R(), W()$ to return the memory locations accessed by the expressions/assignments. This in turn would require a pointer analysis—the very thing we are trying to optimise.

Thus, in the next section, we outline a procedure for computing the relation dep which relies on the symbols (i.e., variable identifiers) occurring in the expressions/assignments rather than the memory locations accessed by them.

Computing dependencies.

In this section we outline how we can compute an over-approximation of the set of assignments A as defined above. Let $\text{symbols}(a)$ be a function that returns the set of variable identifiers occurring in an expression/assignment. For example, $\text{symbols}(a[i] \rightarrow \text{lock}) = \{a, i\}$ and $\text{symbols}(*p = q + 1) = \{p, q\}$. As stated in Sec. 3.1, in our program representation all variable identifiers in a program are unique. We first define the relation sym_2 which indicates whether two assignments have common symbols:

$$\begin{aligned} \text{sym}_2 &\subseteq AS \times AS \\ (a, b) \in \text{sym}_2 &\Leftrightarrow \text{symbols}(a) \cap \text{symbols}(b) \neq \emptyset \end{aligned}$$

Our analysis relies on the following property: If two assignments a, b can access a common memory location (i.e., $M(a) \cap M(b) \neq \emptyset$), then $(a, b) \in \text{sym}_2^*$. This can be seen as follows. Whenever a memory region/location is allocated in C it initially has at most one associated identifier. For example, the memory allocated for a global variable x at program startup has initially just the associated identifier x . Similarly, memory allocated via, e.g., $a = (\text{int} *)\text{malloc}(\text{sizeof}(\text{int}) * \text{NUM})$ has initially only the associated identifier a . If an expression not mentioning x , such as $*p$, can access the associated memory location, then the address of x must have been propagated to p via a sequence of assignments such as $q = \&x, s \rightarrow f = q, p = s \rightarrow f$, with each of

the adjacent assignments having common variables. Thus, if a, b can access a common memory location, then both must be “connected” to the initial identifier associated with the location via such a sequence. Thus, in particular, a, b are also connected. Therefore, $(a, b) \in \text{sym}_2^*$.

We next define the sym relation which also incorporates the starting expressions:

$$\begin{aligned} \text{sym}_1 &\subseteq \text{exprs} \times AS \\ (a, b) \in \text{sym}_1 &\Leftrightarrow \text{symbols}(a) \cap \text{symbols}(b) \neq \emptyset \\ \text{sym} &= \text{sym}_1; \text{sym}_2^* \end{aligned}$$

As we will show below we have $\text{dep} \subseteq \text{sym}$ and thus also $A = \text{dep}|_{(_, a) \rightarrow a} \subseteq \text{sym}|_{(_, a) \rightarrow a}$. Thus, if we compute sym above we get an overapproximation of A .

The fact that $\text{dep} \subseteq \text{sym}$ can be seen as follows. Let $(a, b) \in \text{dep}$. Then there are a_1, a_2, \dots, a_n, b such that $(a_1, a_2) \in \text{dep}_1 \cup \text{dep}_2, (a_2, a_3) \in \text{dep}_2, \dots, (a_n, b) \in \text{dep}_2$. Let (a', a'') be an arbitrary one of those pairs. Then $R(a) \cap W(b) \neq \emptyset$ by the definition of dep_1 and dep_2 . Thus $M(a) \cap M(b) \neq \emptyset$. As we have already argued above, if two expressions/assignments can access the same memory location then they must transitively share symbols. Thus $(a', a'') \in \text{sym}_1 \cup \text{sym}_2^*$ must hold. Therefore, since we have chosen (a', a'') arbitrarily, we have that all of the pairs above are contained in $\text{sym}_1 \cup \text{sym}_2^*$ and thus by the definition of sym and in particular the transitivity of sym_2^* we get $(a, b) \in \text{sym}$.

Thus, we can use the definition of sym above to compute an overapproximation of the set of assignments that can affect the starting expressions as defined semantically in the previous section.

Algorithm.

Algorithm 1 gives our dependency analysis. The first phase (line 1, Algorithm 2) is based on the ideas from the previous section and computes the set of assignments A that can affect the given set of starting expressions exprs . It does so by keeping a global set of variable identifiers I which is initialised to the set of variables occurring in the starting expressions. Then the algorithm repeatedly iterates over the program and checks whether the assignments contain common symbols with I (lines 8–12). If yes, all the symbols in this assignment are also added to I , and the edge is recorded in E . This is repeated until a fixpoint is reached (i.e., I has not changed in an iteration). In addition to assignments, symbols might also be propagated via functions calls and thread creation (from arguments to parameters of the function/thread, and from the return expression to the left-hand side of the call or the argument of the join). This is handled in lines 13–21.

After we have gathered all affecting assignments, the second phase of the algorithm begins (Algorithm 1, lines 2–8). This phase additionally identifies the function calls that might affect the starting expressions. It is based on the following observation. If a function does not contain any affecting assignments, and any of the functions it calls do not either (and those that this function calls in turn do not, etc.), then the calls to the function cannot affect the starting expressions. The ability to prune function calls has a potentially big effect on the performance of the analysis, as it can greatly reduce the amount of code that needs to be analysed.

In the following section we evaluate the performance and effectiveness of the dependency analysis. Its effect on the overall analysis is evaluated in Sec. 7.

Algorithm 1: Dependency analysis

Input : ICFA Prog , lock edges lock_edges
Output: Set of affecting edges A

```

1  $A \leftarrow \text{affecting\_edges}(\text{Prog}, \text{lock\_edges})$ 
2  $F \leftarrow \{f \mid e \in A \wedge f = \text{func}(\text{src}(e))\}$ 
3  $F_h \leftarrow \emptyset$ 
4 while  $F \neq \emptyset$  do
5   remove  $f$  from  $F$ 
6    $F_h \leftarrow F_h \cup \{f\}$ 
7    $E \leftarrow \{e \mid \text{func}(\text{tgt}(e)) = f \wedge$ 
       $\text{op}(e) \in \{\text{func\_entry}, \text{thread\_entry}\}\}$ 
8   for  $e \in E$  do
9      $A \leftarrow A \cup \{e\}$ 
10     $f' \leftarrow \text{func}(\text{src}(e))$ 
11    if  $f' \notin F_h$  then
12       $F \leftarrow F \cup \{f'\}$ 
13 return  $A$ 
```

Evaluation.

We have evaluated the dependency analysis on a subset of 100 benchmarks of the benchmarks given in Sec. 7. For each benchmark the dependency analysis was invoked with the set of starting expressions exprs being those occurring in lock operations or as the first argument of create and join operations. The results are given in the table below.

	runtime	sign. assign.	sign. func.
25th percentile	0.03 s	0.3%	43.3%
arithmetic mean	0.26 s	40.0%	63.3%
75th percentile	0.38 s	72.2%	86.5%

The table shows that the average time (over all benchmarks) to perform the dependency analysis was 0.26 s. The first and last line give the 25th and 75th percentile. This indicates for example that for 25% of the benchmarks it took 0.03 s or less to perform the dependency analysis. The third and fourth column evaluate the effectiveness of the analysis. On average, 40% of the assignments in a program were classified as significant (i.e., potentially affecting the starting expressions). The data also shows that often the number of significant assignments was very low (in 25% of the cases it was 0.3% or less). This happens when the lock usage patterns in the program are simple, such as using simple lock expressions (like `pthread_mutex_lock(&mutex)`) that refer to global locks with simple initialisations (such as using static initialization via `PTHREAD_MUTEX_INITIALIZER`).

The average number of functions classified as significant was 63.3%. This means that on average 36.7% of the functions that occur in a program were identified as irrelevant by the dependency analysis and thus do not need to be analysed by the following pointer analysis.

Overall, the data shows that the analysis is cheap and able to prune a significant number of assignments and functions.

5. NON-CONCURRENCY ANALYSIS

We have implemented an analysis (Algorithm 3) to compute whether two places p_1, p_2 are non-concurrent. That is, the analysis determines whether the statements associated with the places p_1, p_2 (i.e., the operations with which the outgoing edges of $\text{top}(p_1), \text{top}(p_2)$ are labeled) cannot execute concurrently in the contexts embodied by p_1, p_2 .

Whether the places are protected by a common lock is determined by computing the intersection of the must lock-

Algorithm 2: Affecting edges

```

1 function affecting_edges(Prog, lock_edges)
2   A  $\leftarrow$  lock_edges
3   S  $\leftarrow \bigcup_{e \in \text{lock\_edges}} \text{symbols}(\text{op}(e))$ 
4   R  $\leftarrow \emptyset$ 
5   for e  $\in E(\text{Prog})$  do
6     op  $\leftarrow \text{op}(e)$ 
7     if op = (a = b)  $\vee$ 
        op = thread_entry( $\_$ , a, b)  $\vee$ 
        op = func_exit(a, b)  $\vee$ 
        op = thread_join(a, b) then
8       R  $\leftarrow R \cup \{\text{symbols}(a) \cup \text{symbols}(b)\}$ 
9     else if op = func_entry(arg1, ..., argn,
        par1, ..., parn) then
10      R  $\leftarrow R \cup \{\text{symbols}(\text{arg}_i) \cup \text{symbols}(\text{par}_i) \mid$ 
        i  $\in \{1, \dots, n\}\}$ 
11  NM  $\leftarrow$  number_map(R)
12  SM  $\leftarrow$  symbol_map(R)
13  Nh, Sh  $\leftarrow \emptyset, \emptyset$ 
14  while S  $\neq \emptyset$  do
15    remove s from S
16    Sh  $\leftarrow S_h \cup \{s\}$ 
17    for n  $\in SM[s]$  do
18      if n  $\notin N_h$  then
19        Nh  $\leftarrow N_h \cup \{n\}$ 
20        S  $\leftarrow S \cup (NM[r] - S_h)$ 
21  for e  $\in E(\text{Prog})$  do
22    if op = (a = b)  $\vee$ 
        op = func_exit(a, b)  $\vee$ 
        op = thread_join(a, b) then
23      if  $(\text{symbols}(a) \cup \text{symbols}(b)) \cap S_h \neq \emptyset$  then
24        A  $\leftarrow A \cup \{e\}$ 
25  return A

```

sets (lines 3–4). If the intersection is non-empty they cannot execute concurrently and the algorithm returns *true*. Otherwise the algorithm proceeds to check whether the places are non-concurrent due to create and join operations. This is done via a graph search in the ICFA. First the length of the longest common prefix of p_1 and p_2 is determined (line 5). This is the starting point for the ICFA exploration. If there is a path from ℓ_1 to ℓ_2 , it is checked that all the threads that are created to reach place p_1 are joined before location ℓ_2 is reached (and same for a path from ℓ_2 to ℓ_1). This check is performed by the procedure *unwind*(), the full details of which we give in Appendix B.

We evaluated the non-concurrency analysis with respect to what fraction of all the pairs of places p_1, p_2 of a program it classifies as non-concurrent. We found that on a subset of 100 benchmarks of the benchmarks of Sec. 7, it classified 60% of the places corresponding to different threads as non-concurrent on average. We give more data in Appendix B.

6. LOCK GRAPH ANALYSIS

Our lock graph analysis consists of two phases. First, we build a lock graph based on the lockset analysis. In the second phase, we prune cycles that are infeasible due to information from the non-concurrency analysis.

6.1 Lock Graph Construction

Algorithm 3: Non-concurrency analysis

```

Input : places  $p_1, p_2$ , must locksets  $ls_u^1, ls_u^2$ 
Output: true if  $p_1, p_2$  are non-conc., false otherwise
1 if  $p_1 = p_2$  then
2   return true
3 if  $ls_u^1 \cap ls_u^2 \neq \emptyset$  then
4   return true
5 i  $\leftarrow |\text{common\_prefix}(p_1, p_2)|$ 
6  $r_1, r_2 \leftarrow \text{true}, \text{true}$ 
7  $\ell_1, \ell_2 \leftarrow p_1[i], p_2[i]$ 
8 if has_path( $\ell_1, \ell_2$ ) then
9    $r_1 \leftarrow \text{unwind}(i, p_1, \ell_1, \ell_2)$ 
10 if has_path( $\ell_2, \ell_1$ ) then
11    $r_2 \leftarrow \text{unwind}(i, p_2, \ell_2, \ell_1)$ 
12 return  $r_1 \wedge r_2$ 

```

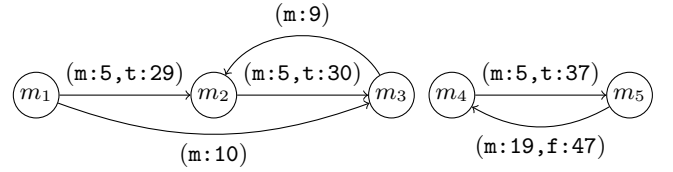


Figure 9: Lock graph for the program in Fig. 4 (t, m and f are shorthand for **thread**, **main** and **func2**, respectively).

A *lock graph* is a directed graph $L \in 2^{Objs^* \times P \times Objs^*}$ (with $Objs^* = Objs \cup \{\star\}$). Each node is a *lock* $\in Objs^*$, and an edge $(lock_1, p, lock_2) \in Objs^* \times P \times Objs^*$ from $lock_1$ to $lock_2$ is labelled with the place p of the lock operation that acquired $lock_2$ while $lock_1$ was owned by the same thread $\text{get_thread}(p)$. Hence, the directed edges indicate the order of lock acquisition. Fig. 9 gives the lock graph for the example program in Fig. 4.

We use the result of the may lockset analysis (Sec. 3.5) to build the lock graph. Fig. 10 gives the lock graph domain that is instantiated in our analysis framework. For each lock operation in place p a thread may acquire a lock $lock_2$ corresponding to the value set of the argument to the lock operation. This happens while the thread may own any lock $lock_1$ in the lockset at that place. Therefore we add an edge $(lock_1, p, lock_2)$ for each pair $(lock_1, lock_2)$.

Finally, we have to handle the indeterminate locks, denoted by \star . We compute the closure $cl(L)$ of the graph w.r.t. edges that involve \star by adding edges from all predecessors of the \star node to all other nodes, and to each successor node of the \star node, we add edges from all other nodes.

6.2 Checking Cycles in the Lock Graph

The final step is to check the cycles in the lock graph. For this purpose we use the information from the non-concurrency analysis. Each cycle c in the lock graph could be a potential deadlock. A cycle c is a set of (distinct) edges; there is a finite number of such sets. A cycle is a potential deadlock if $|c| > 1 \wedge \text{all_concurrent}(c)$ where

$$\begin{aligned}
 \text{all_concurrent}(c) \Leftrightarrow & \\
 & \forall (lock_1, p, lock_2), (lock'_1, p', lock'_2) \in c : \\
 & \neg \text{non_concurrent}(p, p') \vee \\
 & (\text{get_thread}(p) = \text{get_thread}(p')) \wedge \\
 & \text{multiple_thread}(\text{get_thread}(p))
 \end{aligned}$$

and $\text{multiple_thread}(t)$ means that t was created in a loop or

$$\text{Domain: } 2^{Objs^* \times P \times Objs^*}$$

$$s_1 \sqcup s_2 = s_1 \cup s_2$$

With $\text{op}(e) = \text{lock}(a)$:

$$\mathcal{T}[e, p](s) = s \cup \{(lock_1, p, lock_2) \mid \begin{array}{l} lock_1 \in ls_a(p), \\ lock_2 \in vs(p, a) \end{array}\}$$

$$cl(s) = s \cup \{(lock_1, p, lock) \mid \begin{array}{l} (lock_1, p, \star) \in s, \\ lock \in \text{get_locks}(s) \setminus \{lock_1, \star\} \end{array}\}$$

$$\cup \{(lock, p, lock_2) \mid \begin{array}{l} (\star, p, lock_2) \in s, \\ lock \in \text{get_locks}(s) \setminus \{lock_2, \star\} \end{array}\}$$

Figure 10: Lock graph construction

recursion. Due to the use of our non-concurrency analysis we do not require any special treatment for gate locks or thread segments as in [3].

7. EXPERIMENTS

We implemented our deadlock analyser as a pipeline of static analyses in the CPROVER framework,¹ and we performed experiments to support the following hypothesis: *Our analysis handles real-world C code in a precise and efficient way.* We used 715 concurrent C programs that contain locks from the Debian GNU/Linux distribution, with the characteristics shown in Fig. 12.² The table shows that the minimum number of different locks and lock operations encountered by our analysis was 0. We found that this is due to a small number of benchmarks on which the lock operations were not reachable from the main function of the program (i.e., they were contained in dead code).

We additionally selected 8 programs and introduced deadlocks in them. This gives us a benchmark set consisting of 723 benchmarks with a total of 7.1 MLOC. Of these, 715 benchmarks are assumed to be deadlock-free, and 8 benchmarks are known to have deadlocks. The experiments were run on a Xeon X5667 at 3 GHz running Fedora 20 with 64-bit binaries. Memory and CPU time were restricted to 24 GB and 1800 seconds per benchmark, respectively.

Results.

We correctly report (potential) deadlocks in the 8 benchmarks with known deadlocks. The results for the deadlock-free programs are shown in the following table grouped by benchmark size (t/o...timed out, m/o...out of memory):

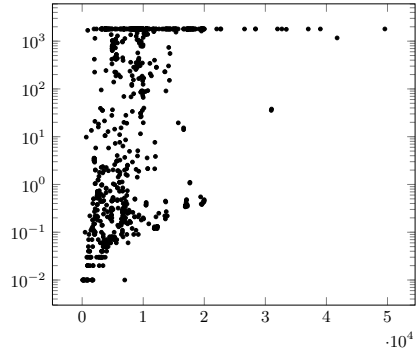
KLOC	analysed	proved	alarms	t/o	m/o
0–5	252	130	55	48	19
5–10	269	91	35	108	35
10–15	86	16	9	54	7
15–20	93	23	5	64	1
20–50	15	2	1	11	1

For 105 deadlock-free benchmarks, we report alarms that are most likely spurious. The main reason for such false alarms is the imprecision of the pointer analysis with respect to dynamically allocated data structures. This leads to lock operations on indeterminate locks (see statistics in Fig. 12). This is a challenging issue to solve, as we discuss in Sec. 10.

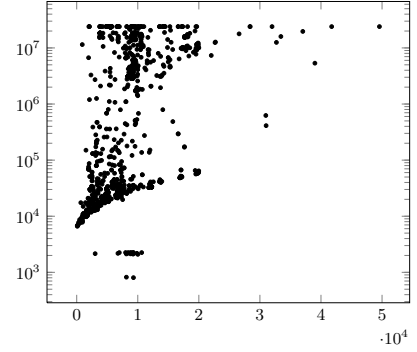
Scatter plots in Figs. 11a and 11b illustrate how the tool scales in terms of running time and memory consumption with respect to the number of lines of code. The tool successfully analyzed programs with up to 40K lines of code.

¹Based on r6268 of <http://www.cprover.org/svn/cbmc/trunk>

²Lines of code were measured using `cloc` 1.53.



(a) LOC vs. user time (timeout 1800 s)



(b) LOC vs. memory consumption (memory limit 24 GB)

Figure 11: Experimental results

As the plots show, the asymptotic behaviour of the algorithms in terms of lines of code is difficult to predict since it mostly depends on the complexity of the pointer analysis.

We evaluated the impact of the different analysis features on a random selection of 83 benchmarks and break down the running times into the different analysis phases on those benchmarks where the tool does not time out or goes out of memory: We found that the dependency analysis is effective at decreasing both the memory consumption and the runtime of the pointer analysis. It decreased the memory consumption by 27% and the runtime by 60% on average. We observed that still the vast majority of the running time (93%) of our tool is spent in the pointer analysis, which is due to the often large number of general memory objects, including all heap and stack objects that may contain locks. May lock analysis (3%), must lock analysis (2%), and lock graph construction (2%) take less time; the run times for the dependency analysis and the cycle checking (up to the first potential deadlock) are negligible. For 80.2% of lock operations the must lock analysis was precise.

Comparison with other tools.

We tried to find other tools to experimentally compare with ours. However, we did not find a tool that handles C code and with which a reasonable comparison could be made. Other tools are either for Java (such as [29]), are based solely on testing (Helgrind [1]), or are semi-automatic lightweight approaches relying on user-supplied annotations (LockLint [2]).

8. THREATS TO VALIDITY

This section discusses the threats to internal and external validity of our results and the strategies we have employed to mitigate them [15].

	max	avg	min
# lines of code	41,749	8,376.5	86
# Threads	163	3.8	1
# Threads in loop	162	2.1	0
# Locks	16	1.5	0
# Lock operations	30773	331.6	0
Precise must analysis	100%	80.2%	0%
Size of largest lockset	8.0	1.2	1.0
# indeterminate locking operations	2106.0	32.3	0.0
# non-concurrency checks	450.0	1.6	0.0

	max	avg	min
Total analysis time (s)	1291.3	435.3	0.0
Dependency analysis	5.0	0.1	0.0
Pointer analysis	1185.1	419.0	0.0
May lockset analysis	345.4	12.5	0.0
Must lockset analysis	29.9	1.1	0.0
Lock graph construction	31.3	1.2	0.0
Cycles detection	327.4	1.4	0.0
Peak memory (GB)	24.0	6.3	0.008

Figure 12: Benchmark characteristics and analysis statistics (for the 367 benchmarks with no time out or out of memory)

The main threat to internal validity concerns the correctness of our implementation. To mitigate this threat we have continually tested our tool during the implementation phase, which has resulted in a testsuite of 122 system-, unit-, and regression-tests. To further test the soundness claim of our tool on larger programs, we introduced deadlocks into 8 previously deadlock-free programs, and checked that our tool correctly detected them. While we have reused existing locks and lock operations to create those deadlocks, they might nevertheless not correspond well to deadlocks inadvertently introduced by programmers.

The threats to external validity concern the generalisability of our results to other benchmarks and programming languages. Our benchmarks have been drawn from a collection of open-source C programs from the Debian GNU/Linux distribution [25] that use pthreads and contain lock operations, from which we ran most of the smaller ones and some larger ones. We found that the benchmark set contains a diverse set of programs. However, we did not evaluate our tool on embedded (safety- or mission-critical) software, a field that we see as a prime application area of a sound tool like ours, due to the ability to verify the absence of errors.

During the experiments we have used a timeout of 1800 s. This in particular means that we cannot say how the tool would fare for users willing to invest more time and computing power; in particular, the false positive rate could increase as programs that take a long time to analyse are likely larger and could have more potential for deadlocks to occur.

Finally, our results might not generalise to other programming languages. For example, while Naik et al. [29] (analysing Java) found that the ability to detect common locks was crucial to lower the false positive rate, we found that it had little effect in our setting, since most programs do not acquire more than 2 locks in a nested manner (see Tab. 12, size of largest lockset).

9. RELATED WORK

Deadlock analysis is an active research area. Since the mid-1990s numerous tools, mainly for C and Java, have been developed. One can distinguish dynamic and static approaches. A common deficiency is that all these tools are neither sound nor complete, and produce false positives and negatives.

Dynamic tools.

The development of the Java PathFinder tool [18, 3] led to ground-breaking work over more than a decade to find lock acquisition hierarchy violation with the help of lock graphs, exposing the issue of gatelocks, and segmentation techniques to handle different threads running in parallel at different times. [4, 21, 33] try to predict deadlocks in executions similar to the observed one. DeadlockFuzzer [21] use a fuzzing technique to search for deadlocking executions. Multicore SDK [27] tries to reduce the size of lock graphs by clustering locks; and Magiclock [10] implements

significant improvements on the cycle detection algorithms. Helgrind [1] is a popular open source dynamic deadlock detection tool, and there are many commercial implementations of dynamic deadlock detection algorithms.

Static tools.

There are very few static deadlock analysis tools for C. LockLint [2] relies on a user-supplied lock acquisition order, and is thus not fully automatic. RacerX [13] focuses more on fast analysis of large code bases than soundness. It performs a path- and context-sensitive analysis, but its pointer analysis is very rudimentary. For Java there is Jlint2 [6], a tool similar to LockLint. The tool Jade [29] consciously uses a may analysis instead of a must analysis, which causes unsoundness. The tools presented in [37] and [35] do not consider gatelock scenarios, which leads to false alarms.

Other tools.

Some tools combine dynamic approaches and constraint solving. For example, CheckMate [20] model-checks a path along an observed execution of a multi-threaded Java program; Sherlock [14] uses concolic testing; and [33, 4] monitor runtime executions of Java programs. There are related techniques to detect synchronisation defects due to blocking communication, e.g. in message passing (MPI) programs [17, 11], for the modelling languages BIP (DFinder tool [7, 8]) or ABS (DECO tool [16]) that use similar techniques based on lock graphs and may-happen-in-parallel information.

Dependency analysis.

Our dependency analysis is related to work on concurrent program slicing [23, 24, 30] and alias analysis [9, 22]. Our analysis is more lightweight than existing approaches as it works on the level of variable identifiers only, as opposed to more complex objects such as program dependence graphs (PDG) or representations of possible memory layouts. Moreover, our analysis disregards expressions occurring in control flow statements (such as if-statements) as these are not relevant to the following pointer analysis which consumes the result of the dependency analysis. The analysis thus does not produce an *executable subset* of the program statements as in the original definition of slicing by Weiser [36].

Non-concurrency analysis.

Our non-concurrency analysis is context-sensitive, works on-demand, and can classify places as non-concurrent based on locksets or create/join. Locksets have been used in a similar way in static data race detection [13], and Havelund [3] used locksets in dynamic deadlock detection to identify non-concurrent lock statements. Our handling of create/join is most closely related to the work of Albert et al. [5]. They consider a language with asynchronous method calls and an await statement that allows to wait for the completion of a previous call. Their analysis works in two phases, the second of which can be performed on-demand, and also provides a form of context-sensitivity. Other approaches, which how-

ever do not work on-demand, include the work of Masticola and Ryder [28] and Naumovich et al. [31] for ADA, and the work of Lee et al. [26] for async-finish parallelism.

10. CONCLUSIONS

We presented a new static deadlock analysis approach for concurrent C/pthreads programs. We demonstrated that our tool can effectively prove deadlock freedom of 2.6 MLOC concurrent C code from the Debian GNU/Linux distribution. Our experiments show that the pointer analysis is the crucial component of a static deadlock analyser. It takes most of the time, by far, and is the primary source for false alarms when the arguments of lock operations cannot be determined. In our evaluation, we observed that the limitations of our pointer analysis regarding dynamically allocated data structures, e.g. when lock objects are stored in lists, are the key reason for false alarms. Future work will focus on addressing this limitation. Moreover, we will integrate the analysis of pthread synchronisation primitives other than mutexes, e.g. condition variables, and extend our algorithm to Java synchronisation constructs. We also want to go beyond lock hierarchy violations, and will include the information from a termination analysis to detect loop-related deadlocks. All this is part of a larger endeavour to show synchronisation correctness and deadlock freedom of all concurrent C programs that use locks (currently 3748 programs with 264.5 MLOC) in the Debian GNU/Linux distribution.

11. ACKNOWLEDGMENTS

This work is supported by ERC project 280053 and SRC task 2269.002.

12. REFERENCES

- [1] <http://valgrind.org/info/tools.html#helgrind>.
- [2] <http://developers.sun.com/solaris/articles/locklint.html>.
- [3] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development*, 54(5):3, 2010.
- [4] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Workshop on Parallel and Distributed Systems: Testing, Analysis*, pages 51–60. ACM, 2006.
- [5] E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of may-happen-in-parallel in concurrent objects. In *FMOODS/FORTE*, pages 35–51, 2012.
- [6] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Australian Software Engineering Conference*, pages 68–75. IEEE, 2001.
- [7] S. Bensalem, M. Bozga, T. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *CAV*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.
- [8] S. Bensalem, A. Griesmayer, A. Legay, T. Nguyen, and D. Peled. Efficient deadlock detection for concurrent systems. In *Formal Methods and Models for Codesign*, pages 119–129. IEEE, 2011.
- [9] M. G. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC*, pages 234–250. Springer, 1995.
- [10] Y. Cai and W. K. Chan. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Trans. Software Eng.*, 40(3):266–281, 2014.
- [11] Z. Chen, X. Li, J. Chen, H. Zhong, and F. Qin. SyncChecker: detecting synchronization errors between MPI applications and libraries. In *International Parallel and Distributed Processing Symposium*, pages 342–353. IEEE, 2012.
- [12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [13] D. R. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles*, pages 237–252. ACM, 2003.
- [14] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Foundations of Software Engineering*, pages 353–365. ACM, 2014.
- [15] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research – an initial survey. In *SEKE*, pages 374–379, 2010.
- [16] A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.
- [17] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *Formal Methods*, volume 8442 of *LNCS*, pages 263–278. Springer, 2014.
- [18] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *SPIN*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.
- [19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
- [20] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Foundations of Software Engineering*, pages 327–336. ACM, 2010.
- [21] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120. ACM, 2009.
- [22] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239. Springer, 2007.
- [23] J. Krinke. Static slicing of threaded programs. In *PASTE*, pages 35–42. ACM, 1998.
- [24] J. Krinke. Context-sensitive slicing of concurrent programs. In *ESEC/FSE*, pages 178–187. ACM, 2003.
- [25] D. Kroening and M. Tautschnig. Automating software analysis at large scale. In *MEMICS*, pages 30–39, 2014.
- [26] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient may happen in parallel analysis for async-finish parallelism. In *SAS*, pages 5–23, 2012.
- [27] Z. D. Luo, R. Das, and Y. Qi. Multicore SDK: A practical and efficient deadlock detector for real-world applications. In *International Conference on Software Testing, Verification and Validation*, pages 309–318. IEEE, 2011.
- [28] S. P. Masticola and B. G. Ryder. Non-concurrency

- analysis. In *PPoPP*, pages 129–138, 1993.
- [29] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering*, pages 386–396. IEEE, 2009.
- [30] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA*, pages 180–190. ACM, 2000.
- [31] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *FSE*, pages 24–34, 1998.
- [32] M. C. Rinard. Analysis of multithreaded programs. In *SAS*, pages 1–19. Springer, 2001.
- [33] F. Sorrentino. PickLock: A deadlock prediction approach under nested locking. In *SPIN*, volume 9232 of *LNCS*, pages 179–199. Springer, 2015.
- [34] U.S.-Canada Power System Outage Task Force. Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations, 2004.
<http://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf>.
- [35] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, 2004.
- [36] M. Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.
- [37] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 602–629. Springer, 2005.

Domain: $\mathcal{D}_i = Fpms \times \mathcal{D}_a$

$$s_i^1 \sqcup_i s_i^2 = s_i^1 \sqcup_s s_i^2$$

With $e = (\ell_1, \ell_2)$, $\text{top}(p) = \text{entry_loc}(\ell_1)$, $f = \text{func}(\ell_2)$, and $n = |p|$:

$\text{next}_i(e, p) =$

$$\begin{cases} \text{entry}_s(p, \ell_2) & \text{op}(e) \in \{\text{thread_entry}, \text{func_entry}\} \\ p[:n-2] + \text{entry_loc}(\ell_2) & \text{op}(e) \in \{\text{func_exit}, \text{thread_exit}, \text{thread_join}\} \\ p[:n-1] & \text{otherwise} \end{cases}$$

$$\mathcal{T}_i[f, p] = \bigcirc_{e \text{ s.t. } P(f, e)} \mathcal{T}_i[e, p]$$

$$\mathcal{T}_i[e, p](s_i) = \mathcal{T}_i[e, p](s_i)$$

Figure 13: Context-, thread-, and flow-insensitive framework

APPENDIX

A. FLOW-INSENSITIVE FRAMEWORK

In the flow-insensitive framework, the topmost location of a place always corresponds to the entry point of a function. That is, we associate a sound overapproximation of the data flow facts that hold for all locations in the function with the entry point of the function. Fig. 13 gives the formalization of the context- and thread-sensitive flow-insensitive framework. It reuses many definitions from the flow-sensitive formalization of Fig. 7.

The result of the flow-insensitive analysis is defined as the least fixpoint of the following equation:

$$s = s_0 \sqcup \lambda p. \mathcal{T}_i[\text{func}(p), p](s(p)) \sqcup_c \bigsqcup_{p', e \text{ s.t. } \text{np}(p, p', e)} \mathcal{T}_i[e, p'](s(p'))$$

$$\begin{aligned} \text{with } \text{np}(p, p', (\ell_1, \ell_2)) = & \text{func}(\ell_1) \neq \text{func}(\ell_2) \\ & \text{entry_loc}(\ell_1) = \text{top}(p') \wedge \\ & \text{entry_loc}(\ell_2) = \text{top}(p) \wedge \\ & \text{next}_i((\ell_1, \ell_2), p') = p \end{aligned}$$

$$\text{with } s \sqcup s' = \lambda p. s(p) \sqcup_c s'(p)$$

B. NON-CONCURRENCY ANALYSIS

We describe how the analysis determines whether two places p_1, p_2 are non-concurrent due to the relationship between threads arising from create and join operations.

The analysis is based on performing a graph search in the ICFA. It makes use of three basic functions on directed graphs. The function `has_path`(ℓ_1, ℓ_2) returns true when there is a path in the ICFA between locations ℓ_1 and ℓ_2 . The function `on_all_paths`(ℓ_1, ℓ_2, ℓ_3) returns true when all paths in the ICFA from ℓ_1 to ℓ_3 pass through ℓ_2 . It is implemented by computing the set of dominators of ℓ_3 (assuming ℓ_1 to be the entry point) and then checking whether ℓ_2 is contained

in that set. The function `in_loop(ℓ)` returns true when there is a path in the ICFA that starts and ends in ℓ .

Algorithm.

We explain the algorithm on an example (Fig. 14). The example consists of four threads (including the main thread). We want to determine whether the statements `x=1` and `x=2` are non-concurrent. We see that they cannot run concurrently as `main()` joins with `thread1()` before starting `thread3()` and `thread1()` joins with `thread2()` before returning.

Let us now look at how our algorithm establishes this fact. The algorithm is called with places $p_1 = (5, 14, 20)$ and $p_2 = (8, 24)$. We have no locks in the example and hence the must locksets are empty (line 3). Line 5 determines the length of the longest common prefix of p_1 and p_2 (which is 0 in this case). This is the starting point for the exploration.

The algorithm then checks whether there is a path from 5 to 8 (line 8). If there would not be a path from either 5 to 8 or 8 to 5 this would mean that 5 and 8 occur in conflicting branches (e.g., one in the then- and the other in the else-branch of an if statement) and thus the places could not be concurrent. In the current case there is a path from 5 to 8.

The algorithm then invokes `unwind()`, which checks that the threads that are created to reach place p_1 are all joined before location 8 is reached. It does so by iterating over p starting from the end. The operation `top(p)` returns the last element of p , and the operation `pop(p)` returns p with the last element removed.

The variable `joined` indicates whether the last thread that was created has been joined yet. If the top element of p corresponds to a create operation (line 10), then if `joined` is `false` the function returns `false`. If not, then `joined` is set to `false`, and the place corresponding to the create is recorded in p_c . Then, a matching join for the create is searched (line 16) by invoking the `find()` function.

The function `find(p_c, p, ℓ_1, ℓ_2)` takes the place p_c , a place p , and locations ℓ_1 and ℓ_2 . The locations ℓ_1 and ℓ_2 are in the same function (let $f = \text{func}(\ell_1)$), and the place p has as top element the call to the function f . The function `find()` looks for a matching join to the create at place p_c . It does so by looking in the function f (lines 3–6), and (recursively) in the callees of f (lines 7–14). The join must occur on all paths between ℓ_1 and ℓ_2 (lines 5, 9). The call `match($p_c, p + \ell_{join}$)` checks that the join at place $p + \ell_{join}$ matches the create at p_c (i.e., the thread ID returned by the `pthread_create()` is the same as the one passed to the `pthread_join()`).

If the creation site is in a loop, we additionally ensure that the thread is joined also on each path that goes back to the same location (lines 17–20). The final lines 21–31 are like the loop body and handle the locations ℓ_1 and ℓ_2 .

For our example, for the first iteration of the while loop in line 4 we have $p = (5, 14, 20)$. In this case, $20 \notin \text{create_locs} \wedge \text{joined}$ in line 7 and we thus continue with the next iteration. Now we have $p = (5, 14)$ and $14 \in \text{create_locs}$ and thus set `joined` to `false` and record $p_c = (5, 14)$. The invocation of `find()` (line 16) finds the join in line 16, and thus `joined` is set to `true`. The while loop then terminates as $|(5)| \leq i + 1$. Lines 21–31 then look for a matching join for the create at location 5. Again such a join is found and `unwind()` returns `true`. The algorithm thus overall returns `true`.

Evaluation.

We have evaluated the non-concurrency analysis on a subset of 100 benchmarks of the benchmarks described in Sec. 7. For each benchmark we randomly selected 1000 pairs of places (p_1, p_2) such that p_1 and p_2 correspond to different

```

12 void *thread1() {
13     pthread_t tid2;
14     pthread_create(
15         &tid2, 0, thread2, 0);
16     pthread_join(tid2, 0);
17     return 0;
18 }
19 void *thread2() {
20     x = 1;
21     return 0;
22 }
23 void *thread3() {
24     x = 2;
25     return 0;
26 }
1 int main()
2 {
3     pthread_t tid1;
4     pthread_t tid3;
5     pthread_create(
6         &tid1, 0, thread1, 0);
7     pthread_join(tid1);
8     pthread_create(
9         &tid3, 0, thread3, 0);
10    return 0;
11 }

```

Figure 14: Statements `x=1` and `x=2` are non-concurrent

Domain: $2^{Objs} \cup \{\star\}$
$s_1 \sqcup s_2 = s_1 \cap s_2$
With $\text{op}(e) = \text{lock}(a)$:
$\mathcal{T}[e, p](s) = \begin{cases} s \cup \text{vs}(p, a) & \text{if } \text{vs}(p, a) = 1 \wedge \text{vs}(p, a) \neq \{\star\} \\ s & \text{otherwise} \end{cases}$
With $\text{op}(e) = \text{unlock}(a)$:
$\mathcal{T}[e, p](s) = \begin{cases} s - \text{vs}(p, a) & \text{if } \text{vs}(p, a) \neq \{\star\} \\ \emptyset & \text{otherwise} \end{cases}$
With $\text{op}(e) \in \{\text{thread_entry}, \text{thread_exit}, \text{thread_join}\}$:
$\mathcal{T}[e, p](s) = \emptyset$

Figure 15: Must lockset analysis

threads. We then performed the non-concurrency check for each of the 1000 pairs. The results are given in the table below.

	runtime	n.c. places	n.c. lock places
25th percentile	0.14s	47%	11%
arithmetic mean	4.07s	60%	43%
75th percentile	4.56s	79%	67%

The table shows that the average time (over all benchmarks) it took to perform 1000 non-concurrency checks was 4.07s. The first and last line give the 25th and 75th percentile. This indicates for example that for 25% of the benchmarks it took 0.14s or less to perform 1000 non-concurrency checks. The third and fourth column evaluate the effectiveness of the non-concurrency analysis. The third column shows that on average our analysis classified 60% of the place pairs as non-concurrent. The fourth column gives the same property while only regarding places that correspond to lock operations. The number of places classified as non-concurrent is lower in this case, which is expected as the code portions using locks are those that can run concurrently with others. Overall, the data shows that the non-concurrency analysis is both fast and effective.

C. MUST LOCKSET ANALYSIS

Fig. 15 gives a formalisation of the must lockset analysis. The must locksets can never contain the value \star .

D. FRAMEWORK IMPLEMENTATION

Algorithm 4: Find join

```
1 function find( $p_c, p, \ell_1, \ell_2$ )
2    $f \leftarrow \text{func}(\ell_1)$ 
3    $\text{join\_locs} \leftarrow \{\ell \in L(f) \mid \exists e = (\ell, \_): \text{is\_join}(\text{op}(e))\}$ 
4   foreach  $\ell_{\text{join}} \in \text{join\_locs}$  do
5     if  $\text{on\_all\_paths}(\ell_1, \ell_{\text{join}}, \ell_2) \wedge \text{match}(p_c, p + \ell_{\text{join}})$ 
6       then
7         return true
8    $\text{entry\_edges} \leftarrow \{e = (\ell_{\text{src}}, \ell_{\text{tgt}}) \mid \text{func}(\ell_{\text{src}}) =$ 
9      $f \wedge \text{op}(e) = \text{func\_entry}\}$ 
10  foreach  $(\ell_{\text{src}}, \ell_{\text{tgt}}) \in \text{entry\_edges}$  do
11    if  $\text{on\_all\_paths}(\ell_1, \ell_{\text{src}}, \ell_2)$  then
12       $p' \leftarrow p + \ell_{\text{src}}$ 
13       $f' \leftarrow \text{func}(\ell_{\text{tgt}})$ 
14       $r \leftarrow \text{find}($ 
15         $p_c,$ 
16         $p',$ 
17         $\ell_{\text{tgt}},$ 
18         $\text{exit\_loc}(f'))$ 
19      if  $r$  then
20        return true
21  return false
```

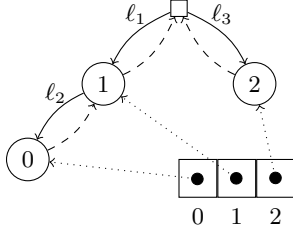


Figure 16: Trie- and array-based place map data structure, representing the two way mapping $(\ell_1) \leftrightarrow 1$, $(\ell_1, \ell_2) \leftrightarrow 0$, $\ell_3 \leftrightarrow 2$

Fig. 16 gives an example of a place map which contains the mappings $(\ell_1) \leftrightarrow 1$, $(\ell_1, \ell_2) \leftrightarrow 0$, and $(\ell_3) \leftrightarrow 2$. Unlike in an ordinary trie, in our implementation the nodes also have pointers to their parent (dashed arrows). This allows to reconstruct a place from a pointer to a node in the trie. For example, by starting from the leaf node labeled with 0 we can traverse the parent edges backwards to get the place (ℓ_1, ℓ_2) .

E. CORRECTNESS PROOFS

In this section, we show the correctness of our deadlock analysis approach. We show that the may locksets computed for each place overapproximate the sets of locks a thread may hold in a concrete execution at that place. Based on the correctness of the may locksets, we then show that our lock graph overapproximates the potential lock allocation graphs (LAGs). That is, we show that for each execution prefix of the program for which there exists a cycle in the LAG, there also is a cycle in the lock graph. We assume the correctness of the pointer analysis and the correctness of the non-concurrency analysis.

E.1 Preliminaries

In Sec. 3.3 we have formulated our analyses as a fixpoint

Algorithm 5: Unwind

```
1 function unwind( $i, p, \ell_1, \ell_2$ )
2    $\text{create\_locs} \leftarrow \{\ell \mid \exists e = (\ell, \_): \text{op}(e) = \text{thread\_entry}\}$ 
3    $\text{joined} \leftarrow \text{true}$ 
4   while  $|p| > i + 1$  do
5      $\ell \leftarrow \text{top}(p)$ 
6      $f \leftarrow \text{func}(\ell)$ 
7     if  $\ell \notin \text{create\_locs} \wedge \text{joined}$  then
8        $p \leftarrow \text{pop}(p)$ 
9       continue
10    if  $\ell \in \text{create\_locs}$  then
11      if  $\neg \text{joined}$  then
12        return false
13       $\text{joined} \leftarrow \text{false}$ 
14       $p_c \leftarrow p$ 
15       $p \leftarrow \text{pop}(p)$ 
16       $\text{joined} \leftarrow \text{find}(p_c, p, \ell, \text{exit\_loc}(f))$ 
17      if  $\text{in\_loop}(\ell)$  then
18         $\text{loop\_joined} \leftarrow \text{find}(p_c, p, \ell, \ell)$ 
19        if  $\neg \text{joined} \vee \neg \text{loop\_joined}$  then
20          return false
21  if  $\ell_1 \in \text{create\_locs}$  then
22    if  $\neg \text{joined}$  then
23      return false
24     $\text{joined} \leftarrow \text{false}$ 
25     $p_c \leftarrow p$ 
26  if  $\neg \text{joined}$  then
27     $p \leftarrow \text{pop}(p)$ 
28     $\text{joined} \leftarrow \text{find}(p_c, p, \ell_1, \ell_2)$ 
29    if  $\text{in\_loop}(\ell_1)$  then
30       $\text{loop\_joined} \leftarrow \text{find}(p_c, p, \ell_1, \ell_1)$ 
31      return  $\text{joined} \wedge \text{loop\_joined}$ 
32  return joined
```

computation over the ICFA. An analysis computes data flow facts for each *place* (see the fixpoint equation in Sec. 3.3). The analysis can also be viewed in a slightly different way: as computing a fixpoint over a larger structure that we term *context-sensitive control-flow automaton* (CCFA). The CCFA for a program is just like the ICFA, but with the nodes being places rather than locations. Two places p, p' are connected by an edge if $\text{top}(p)$ and $\text{top}(p')$ are connected by an edge in the corresponding ICFA.

We denote a concrete *execution* (or execution prefix) of a program as an interleaving $E = (p_1, t_1, o_1)(p_2, t_2, o_2) \dots (p_n, t_n, o_n)$. The p_i are places, the t_i are concrete thread IDs, and the o_i are execution instances of the operations with which the edges connecting the places p_i, p_{i+1} are labeled. We refer to the individual tuples that make up E as *steps*. We use array subscript notation (0-based) to refer to individual elements of lists and tuples. For example, $E[0][2]$ refers to the third component of the first tuple of execution E . We further use slice notation to refer to contiguous subsequences of executions. We further use slice notation (e.g., $E[n:m]$) for contiguous subsequences of executions (see Sec. 3.2).

We denote an *execution path* (or an execution path prefix) of a program as $E_p = (p_1, op_1)(p_2, op_2) \dots (p_n, op_n)$. An execution path of a program is a path through its CCFA, starting at the entry point. Here the op_i are the operations with which the edges connecting the places are labeled (rather than concrete instances of those operations).

Given a thread ID t (resp. abstract thread ID t'), we denote by $E|t$ (resp. $E_p|t'$) the executions (resp. execution paths) projected to the steps of the given thread t (resp. abstract thread t'). We further denote by $T(E) = \{t \mid (-, t, -) \in E\}$ the set of threads in execution E .³ A thread always starts with a `thread_entry` operation, thus we have $(E|t)[0][2] = \text{thread_entry}(\dots)$ and $(E_p|t')[0][1] = \text{thread_entry}(\dots)$.

Property 1. Let E be an execution prefix and let $E' = E|t$ ($n = |E'|$) be the execution of some thread t in E . Then, there is an execution path prefix E_p ($m = |E_p|$) with

$$E_p[m - n :]|_{(p, -) \mapsto p} = E'|_{(p, -) \mapsto p}$$

The property holds by the shape of the CCFA (which directly derives from the shape of the ICFA). The property states that for each execution E and thread t in E , there is an execution path prefix E_p such that the sequence of places visited by t and the sequence of places visited by the suffix of E_p are the same.

We next define a concretisation function which states how the result of the pointer analysis (and hence also the static locksets) are interpreted.

Definition 1. Let A be the set of all locks that may be held in any execution of a program. Then:

$$c(ls) = \begin{cases} A & ls = \{\star\} \\ ls & \text{otherwise} \end{cases}$$

The function satisfies the property $c(ls_1) \cup c(ls_2) = c(ls_1 \cup ls_2)$. We are now in a position to prove the correctness of the may lockset analysis.

E.2 May lockset correctness

If $E = (p_1, t_1, o_1) \dots (p_n, t_n, o_n)$ is an execution prefix we denote by $ls_c(E)$ the concrete set of locks before executing the final step (p_n, t_n, o_n) . This is the set of locks held by

³We use the same notation (\in) to denote elements of sets and lists.

thread t_n at that step. A thread starts with an empty set of locks held.

Theorem E.1. Let $Prog$ be a program and let $E = (p_1, t_1, o_1) \dots (p_n, t_n, o_n)$ be an execution prefix of $Prog$. Then:

$$ls_c(E) \subseteq c(ls_a(p_n)).$$

PROOF. Let $t = t_n$ and let $E' = E|t$. By Property 1 there is an execution path E_p that ends in a sequence E'_p which consists of the same sequence of places as E' . We write $ls_c(i)$ for the concrete lockset before executing step i of E' . These locksets result from the execution of the program. We write $ls_a(i)$ for the may lockset before handling step i of E'_p . These locksets are the result of applying the transfer function defined in Fig. 8.

We next show by induction that the may locksets computed by our analysis for each step along E'_p overapproximate the concrete locksets of E' at the corresponding steps. That is, we show that for all $0 \leq i < |E'|$: $ls_c(i) \subseteq c(ls_a(i))$. Each thread starts with an empty lockset. In our analysis this is reflected by the clause (2) in Fig. 8. Hence the base case $ls_c(0) = \emptyset \subseteq \emptyset = c(\emptyset) = c(ls_a(0))$ holds.

We next show the induction step via a case distinction based on whether step i is (1) a lock operation or (2) an unlock operation.

(1) Let $ls_c(i) \subseteq c(ls_a(i))$ and let $p = E'[i][0]$. We show that then also $ls_c(i+1) \subseteq c(ls_a(i+1))$ after a lock operation $\text{lock}(a)$. We perform a case distinction over the cases of the definition of $\mathcal{T}[\cdot](ls_a(i))$ (see Fig. 8).

(Case 1) Let l be the concrete lock acquired. By the correctness of the pointer analysis, we have $\{l\} \subseteq c(vs(p, a))$. Therefore:

$$ls_c(i+1) = ls_c(i) \cup \{l\} \subseteq c(ls_a(i)) \cup c(vs(p, a)) \subseteq c(ls_a(i) \cup vs(p, a)) = c(ls_a(i+1))$$

(Case 2) $ls_c(i+1) \subseteq A = c(\{\star\})$ holds since A is the set of all locks.

(2) Let $ls_c(i) \subseteq c(ls_a(i))$ and let $p = E'[i][0]$. We show that then also $ls_c(i+1) \subseteq c(ls_a(i+1))$ after an unlock operation $\text{unlock}(a)$. We perform a case distinction over the cases of the definition of $\mathcal{T}[\cdot](ls_a(i))$ (see Fig. 8).

(Case 1) Since $|ls_a(i)| = 1$ and $ls_a(i) \neq \{\star\}$, we also have $|ls_c(i)| = 1$. Therefore, $ls_c(i+1) = \emptyset \subseteq \emptyset = c(\emptyset) = c(ls_a(i+1))$.

(Case 2) Let l be the concrete lock released. Thus we have $l \in c(ls_a(i))$ and $l \in c(vs(p, a))$. Since $|ls_a(i) \cap vs(p, a)| = 1$ we have that $c(ls_a(i) \cap vs(p, a)) = \{l\}$. Thus, $c(ls_a(i) - vs(p, a)) = c(ls_a(i)) - \{l\}$. Therefore, $ls_c(i+1) = ls_c(i) - \{l\} \subseteq c(ls_a(i)) - \{l\} = c(ls_a(i) - vs(p, a)) = c(ls_a(i+1))$.

(Case 3) $ls_c(i+1) \subseteq ls_c(i)$ and thus $ls_c(i+1) \subseteq c(ls_a(i)) = c(ls_a(i+1))$.

Thus, we have shown that the may lockset is an overapproximation of the concrete lockset at any step along E'_p , and thus in particular also at the final step of E'_p (i.e., at the place associated with the final step of E'_p). We can now use the properties of data flow analyses to complete the proof. First, since the may lockset computed for the final place of E_p is an overapproximation of the concrete lockset, it follows that the “meet over all paths” (MOP) at this place is an overapproximation of the concrete locksets for all concrete executions that might reach that place.

Second, the analysis given in Fig. 8 consists of a finite lattice with top element $\{\star\}$, join function \sqcup , and a monotonic transfer function. Consequently, the minimal fixpoint solution (MFP) of the data flow equations overapproximates the MOP solution. Therefore, since the MOP solution is sound, the MFP solution is also sound. \square

E.3 Lock graph correctness

During a concrete execution of a program, each step of the execution has an associated *lock allocation graph* (LAG). The LAG has two types of nodes: threads and locks. There is an edge from a lock node to a thread node if the lock is assigned to that thread (allocation edge). There is an edge from a thread node to a lock node if the thread has requested the lock (a request edge). If the LAG at a certain step in the execution has a cycle, then the involved threads have deadlocked. Those threads cannot make any more steps (but other threads might). We thus need to show that whenever there is an execution that has a cyclic LAG, then our lock graph also has a cycle c' for which $\text{all_concurrent}(c')$ holds.

We first show a lemma about the lock graph closure computation (see Fig. 10) that we will use later on.

Lemma E.2. Let ls_1, ls_2, ls_3, ls_4 be nonempty static locksets, and let p, p' be places. Let further $l \in c(ls_2)$ and $l \in c(ls_3)$. Let $L = \{(lock_1, p'', lock_2) \mid (lock_1 \in ls_1 \wedge lock_2 \in ls_2 \wedge p'' = p) \vee (lock_1 \in ls_3 \wedge lock_2 \in ls_4 \wedge p'' = p')\}$. Then:

$$\begin{aligned} \forall lock_1 \in ls_1, lock_2 \in ls_4: \exists lock: \\ (lock_1, p, lock), (lock, p', lock_2) \in cl(L) \end{aligned}$$

PROOF.

- (1) Assume $lock \in ls_2, lock \in ls_3$ ($lock$ may be \star). Then, by the definition of L above, for all locks $lock_1 \in ls_1, lock_2 \in ls_4$, $(lock_1, p, lock), (lock, p', lock_2) \in L \subseteq cl(L)$.
- (2) Assume $ls_2 = \{\star\}, lock \in ls_3, lock \neq \star$. Then for all locks $lock_1 \in ls_1, lock_2 \in ls_4, (lock_1, p, \star), (lock, p', lock_2) \in L$. Then in $cl(L)$ there is an edge $(lock_1, p, lock)$ by the definition of $cl()$.
- (3) Assume $lock \in ls_2, lock \neq \star, ls_3 = \{\star\}$. This case is symmetric to (2). \square

We next show a lemma about the definition of $\text{all_concurrent}()$.

Lemma E.3. Let E be an execution prefix, let G be the LAG at its final step, and let c be a cycle in G . Let t_1, \dots, t_n be the threads involved in the cycle c , and let $(p_1, t_1, o_1), \dots, (p_n, t_n, o_n)$ be last steps of each thread involved in c in E . Then for all p_i, p_j :

$$\begin{aligned} \neg \text{non_concurrent}(p_i, p_j) \vee \\ (\text{get_thread}(p_i) = \text{get_thread}(p_j) \wedge \text{multiple_thread}(p_i)) \end{aligned}$$

PROOF. Since in E all steps $(p_1, t_1, o_1), \dots, (p_n, t_n, o_n)$ could reach a lock operation on which they blocked, they must have been able to run concurrently in this execution. Now for two places p_i, p_j , $\text{get_thread}(p_i) \neq \text{get_thread}(p_j)$, it follows that $\neg \text{non_concurrent}(p_i, p_j)$ by the correctness of the non-concurrency analysis.

Now assume $\text{get_thread}(p_i) = \text{get_thread}(p_j)$. In this case the places have the same abstract thread ID but they occur in *different* concrete threads. This occurs when a thread create operation occurs in a loop or a recursion (or the call to the function that invokes the thread creation operation occurs in a loop or recursion, etc.). Then we have $\text{multiple_thread}(p_i)$. \square

We can now show the main theorem stating the soundness of our lock graph and cycle search.

Theorem E.4. Let $Prog$ be a program and let $E = (p_1, t_1, o_1) \dots (p_n, t_n, o_n)$ be an execution prefix of $Prog$. Then if the LAG at the final step of E has a cycle, then the lock graph of $Prog$ has a cycle c' with $\text{all_concurrent}(c')$.

PROOF. Let c denote a cycle in the LAG at the final step of E . In this cycle, every thread and every lock occurs exactly once. Let t, t' be two threads involved in the cycle such that there are edges $(l_1, t), (t, l_2), (l_2, t'), (t', l_3)$, for locks l_1, l_2, l_3 (we might have that $l_1 = l_3$).

Let $n = |E|t|$, $m = |E|t'|$, and let $(p, t, o) = (E|t)[n - 1]$, $(p', t', o') = (E|t')[m - 1]$ be the last steps of $E|t, E|t'$. The steps o, o' are lock operations, i.e., $o = \text{lock}(exp_1 : l_2)$, $o' = \text{lock}(exp_2 : l_4)$ with expression exp_1 referring to l_2 and expression exp_2 referring to l_4 . That is, l_2 is the lock requested by o , and l_4 is the lock requested by o' . Moreover, l_1 is in the lockset ls_c at the last step of $E|t$, and l_2 is in the lockset ls'_c at the last step of $E|t'$.

The analysis visits each place at least once, hence the transfer function (see Fig. 10) is also applied to the edges outgoing from the places p and p' . Applying the transfer function to the lock edge starting at p adds edges from the elements of $ls_a(p)$ to the elements of $vs(p, exp_1)$ to the lock graph L . Applying the transfer function to the lock edge starting at p' adds edges from the elements of $ls_a(p')$ to the elements of $vs(p', exp_2)$ to the lock graph L . By the correctness of the may lockset analysis and the correctness of the pointer analysis we have $l_2 \in c(vs(p, exp_1)), l_2 \in c(ls_a(p'))$. Therefore, by Lemma E.2, it follows that for all locks $lock_1 \in ls_a(p), lock_2 \in vs(p', exp_2)$, there is a lock $lock$ such that $(lock_1, p, lock), (lock, p', lock_2) \in L$.

Thus, in the previous paragraph, we have shown that for any portion of the cycle c consisting of adjacent threads t, t' and edges $(l_1, t), (t, l_2), (l_2, t'), (t', l_3)$, there is a portion $(lock_1, p, lock), (lock, p', lock_2)$ in the lock graph. Therefore, the lock graph also has a cycle c' . The places p, p' are the places associated with the final steps of the threads in E that are involved in the cycle c in the LAG. Hence, by Lemma E.3, it follows that $\text{all_concurrent}(c')$. \square

This figure "locs_mem_deadlock.png" is available in "png" format from:

<http://arxiv.org/ps/1607.06927v1>

This figure "locs_time_deadlock.png" is available in "png" format from:

<http://arxiv.org/ps/1607.06927v1>